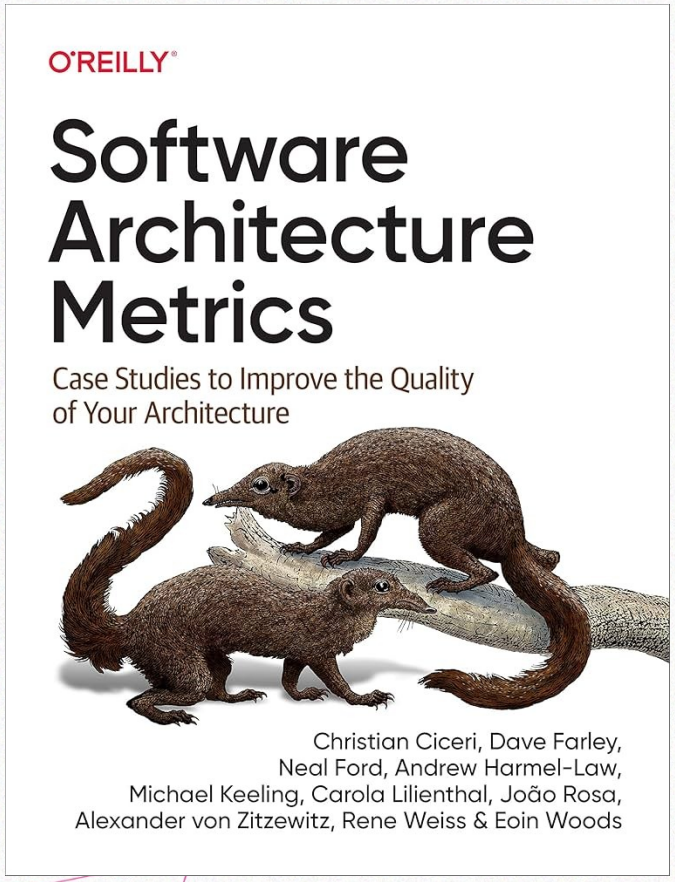
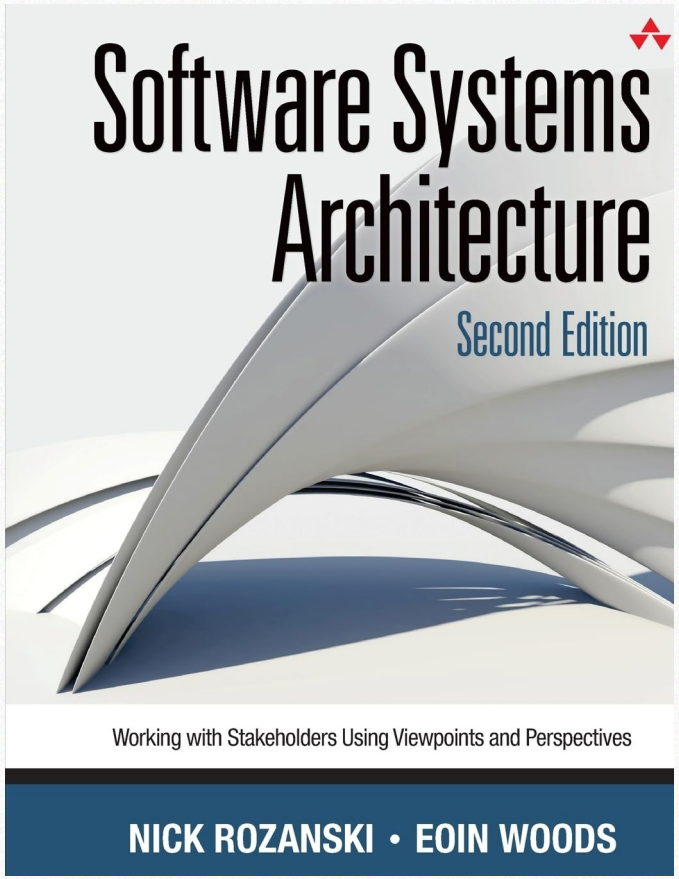
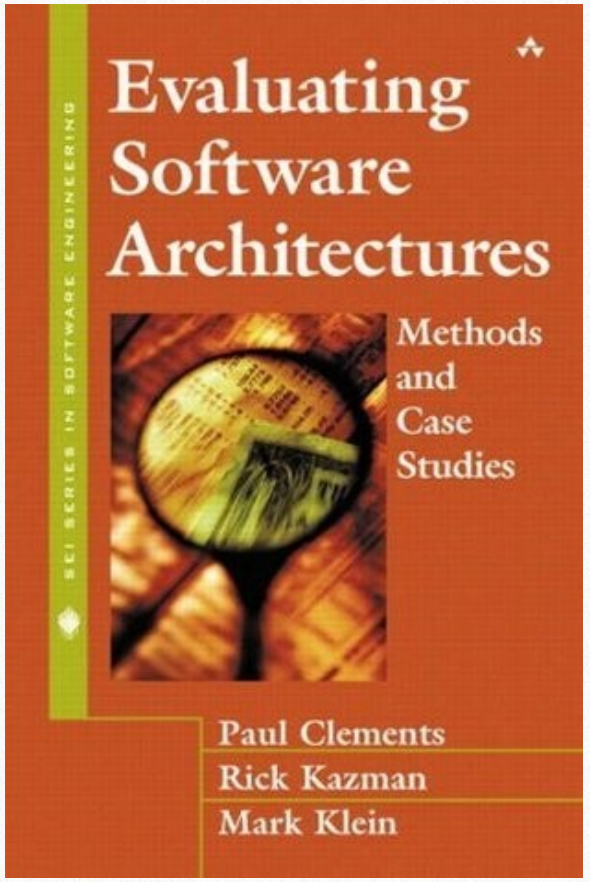
Observabilidad y al Monitoreo

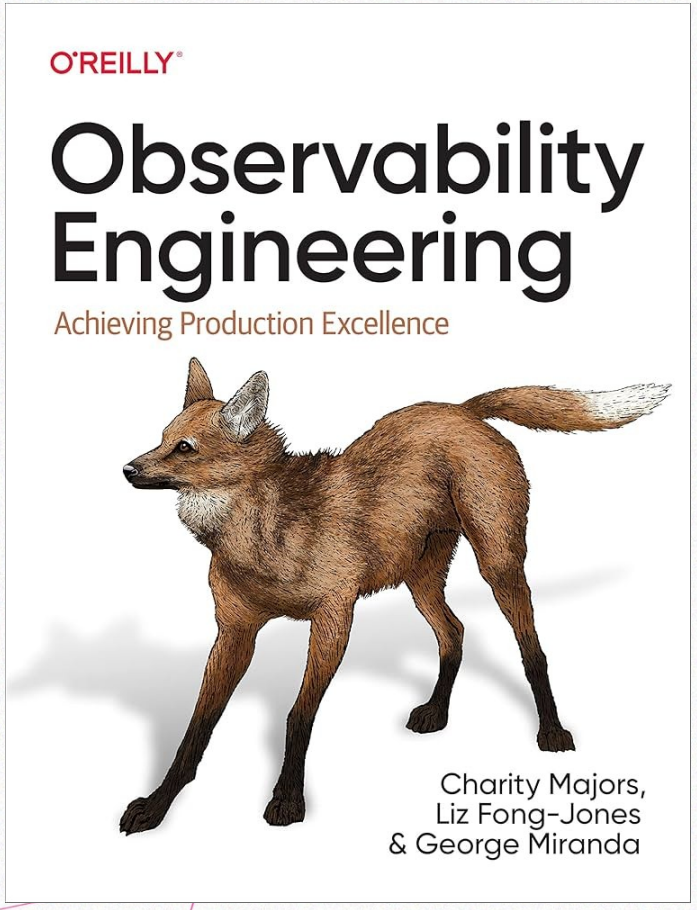
“El diseño arquitectónico de software es la clave para construir sistemas sostenibles.”

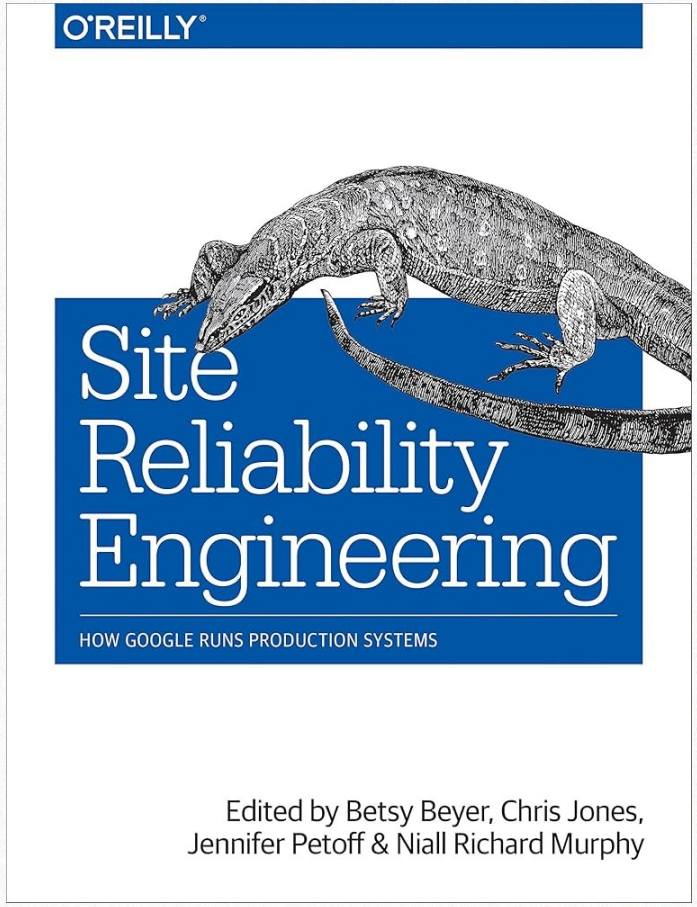
— Grady Booch

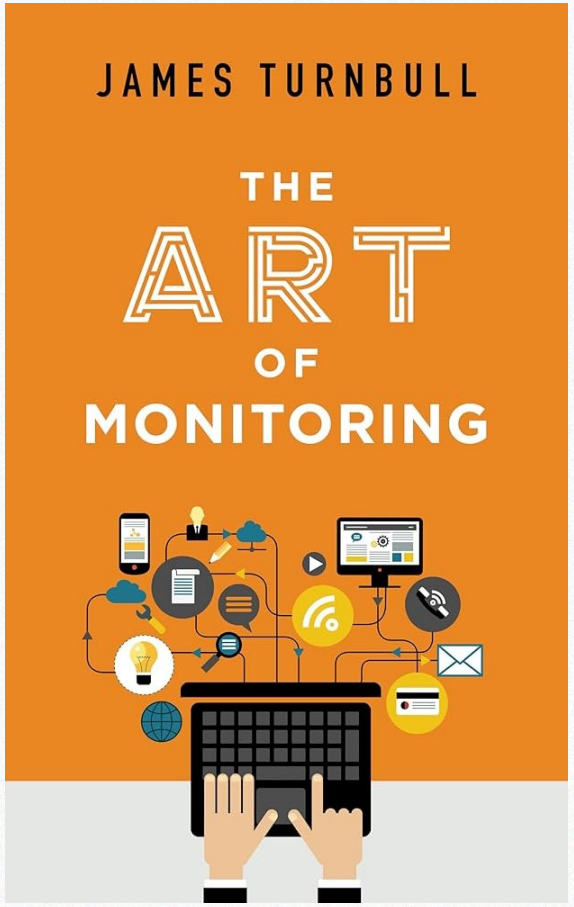

O'REILLY
Software Architecture Metrics
Case Studies to Improve the Quality of Your Architecture
Christian Ciceri, Dave Farley, Neal Ford, Andrew Harmel-Law, Michael Keeling, Carola Lilienthal, João Rosa, Alexander von Zitzewitz, Rene Weiss & Eoin Woods


Software Systems Architecture
Second Edition
Working with Stakeholders Using Viewpoints and Perspectives
NICK ROZANSKI • EOIN WOODS


SEI SERIES IN SOFTWARE ENGINEERING
Evaluating Software Architectures
Methods and Case Studies
Paul Clements
Rick Kazman
Mark Klein


O'REILLY
Observability Engineering
Achieving Production Excellence
Charity Majors,
Liz Fong-Jones
& George Miranda


O'REILLY
Site Reliability Engineering
HOW GOOGLE RUNS PRODUCTION SYSTEMS
Edited by Betsy Beyer, Chris Jones,
Jennifer Petoff & Niall Richard Murphy


JAMES TURNBULL
THE
ART
OF
MONITORING

¿Qué es lo último que monitorizaron y que resultó inútil cuando hubo un problema real?

¿Por qué es relevante para la Arquitectura de Software?

Rozanski y Woods (2012), en su obra *Software Systems Architecture*, mencionan la importancia de contemplar los puntos de vista operativos y de soporte para asegurar la calidad del sistema en su entorno real.

- **Confiabilidad**: Un sistema con buena observabilidad facilita la detección y resolución de problemas, aumentando la confianza de los usuarios.
- **Escalabilidad**: Monitorear y observar el comportamiento del sistema bajo carga o ante picos de demanda permite tomar decisiones de escalamiento oportunas.
- **Mantenibilidad**: Con herramientas y prácticas de observabilidad bien diseñadas, los equipos de desarrollo y operaciones pueden mantener y evolucionar el sistema de manera más efectiva.

"¿Por qué estos conceptos son tan importantes en el contexto de la arquitectura de software moderna?"

Definiciones y Diferencias Clave

Monitoreo

Es el conjunto de prácticas y herramientas destinadas a recopilar y analizar datos sobre un sistema para comprender su estado y rendimiento en tiempo real.
De manera tradicional, se centra en:

- **Métricas conocidas**: CPU, memoria, uso de disco, latencia de red, etc.
- **Umbrales de alerta**: Se configuran alertas cuando una métrica excede un límite predefinido (por ejemplo, uso de CPU > 80%).
- **Paneles de control (dashboards)**: Herramientas como Grafana, Prometheus o Nagios permiten visualizar datos en tiempo real.

Definiciones y Diferencias Clave

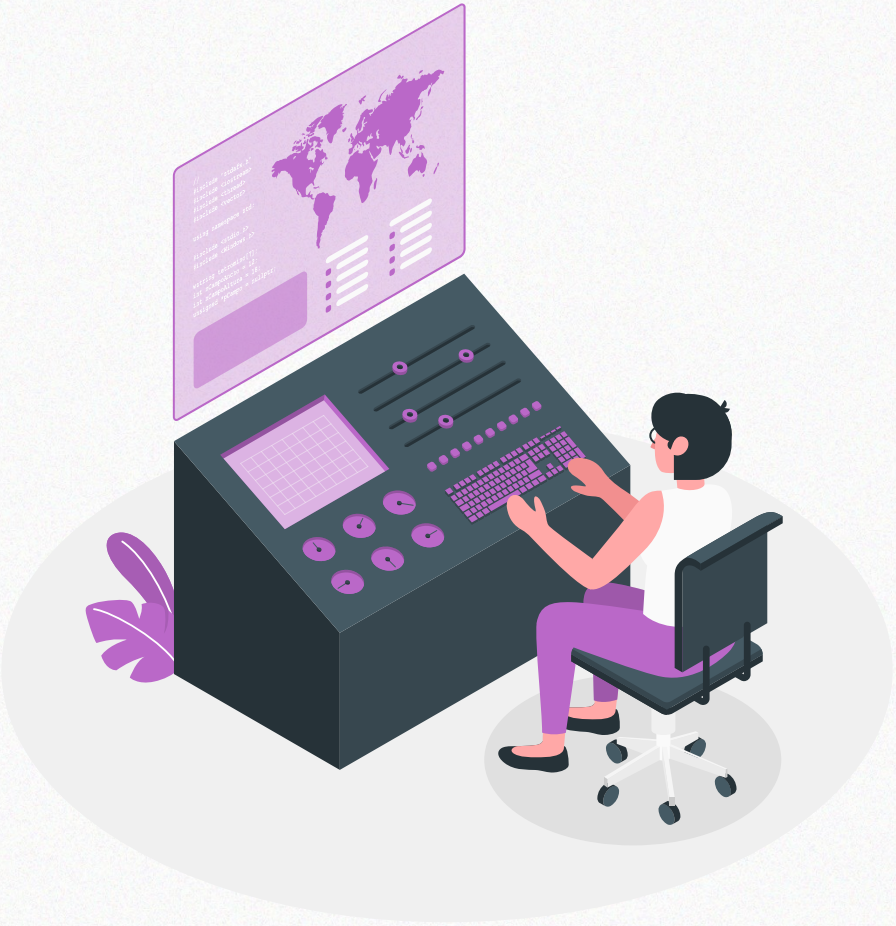
Desconstrucción del concepto de monitoreo

Cuando diseñan el monitoreo
¿Cómo deciden qué medir?

El monitoreo parte de suposiciones previas: sé qué quiero medir, sé qué problemas pueden pasar.

Si falla algo fuera de esas métricas → queda invisible.

Imaginen que algo falla en un servicio que nunca habían pensado que podía romperse. ¿Podrían, con las señales actuales, reconstruir qué pasó?



Definiciones y Diferencias Clave

Observabilidad

La capacidad de responder preguntas nuevas e imprevistas sobre tu sistema, usando solo la información que este expone

-- Charity Majors

¿Qué necesitarían para reconstruir la secuencia exacta de eventos de ese fallo desconocido?

- Logs detallados, trazas, métricas correlacionadas, contexto de ejecución.

¿Eso implica que ya sabíamos qué métricas medir?

- No, necesitamos datos ricos que sirvan para muchas preguntas distintas.

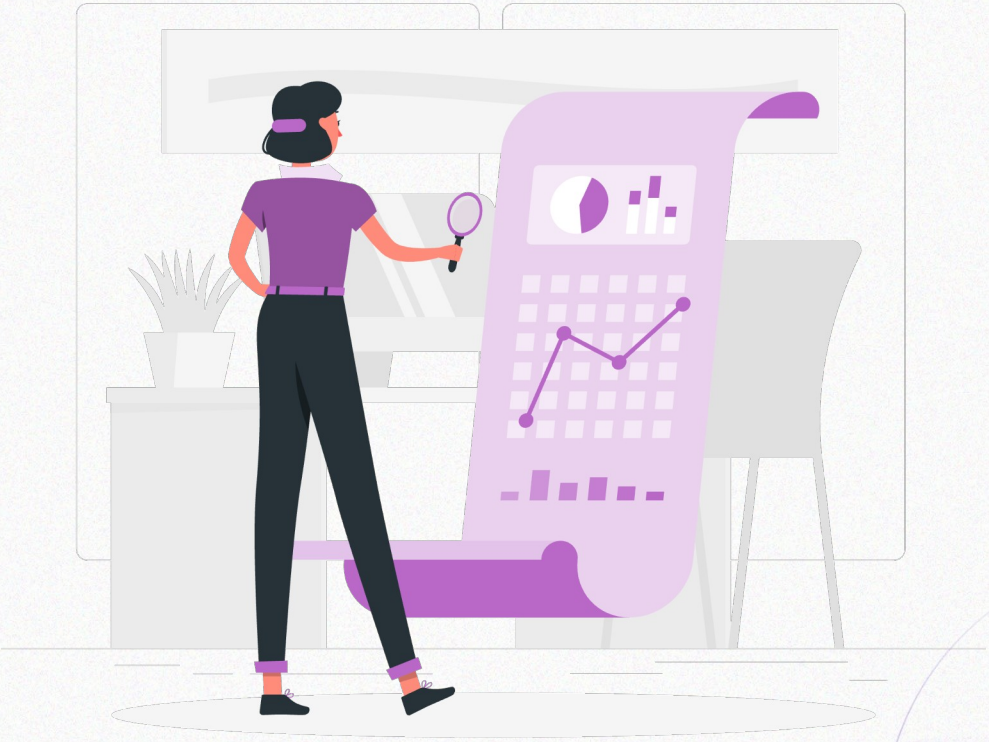
Áreas clave de Observability Engineering

1. Instrumentación por diseño
“Qué es más barato, instrumentar desde el inicio o añadirlo en producción?”
Ejemplo: OpenTelemetry embebido desde la primera release.

2. Curación de señales (signal curation)
“Qué hacen con las métricas o logs que ya no sirven?”
Ejemplo: eliminar métricas obsoletas para evitar ruido y costo.

3. Correlación y exploración
“Cuánto tiempo tardan hoy en unir datos de logs, métricas y trazas?”
Ejemplo: unificar contexto en una herramienta tipo Honeycomb o Jaeger.

4. Prácticas de SLOs y SLIs
“Qué pasa si tu SLO está mal definido?”
Ejemplo: medir disponibilidad en % de uptime vs. medir éxito de transacciones completadas.



Reflexion

¿Cuáles serían las métricas más importantes para monitorear en un sistema de comercio electrónico que recibe miles de transacciones por minuto?

Ejemplo

Imaginemos que tenemos un servicio de comercio electrónico en el que los usuarios realizan compras. Para monitorear, colocaríamos un “contador” de transacciones procesadas por minuto y un registro de la latencia promedio.

Si la latencia sube por encima de 300 ms, se desencadena una alerta.

Observabilidad

Es la capacidad de un sistema para exponer suficiente información interna que permita entender por qué el sistema se comporta de determinada forma, incluso ante situaciones desconocidas o imprevistas

1. **Logs**: Registros detallados de eventos.
2. **Métricas**: Datos cuantificables (velocidad de respuesta, porcentaje de errores, uso de recursos).
3. **Trazas (Tracing)**: Rastrean la ruta de una petición a través de múltiples servicios, mostrando el recorrido paso a paso.

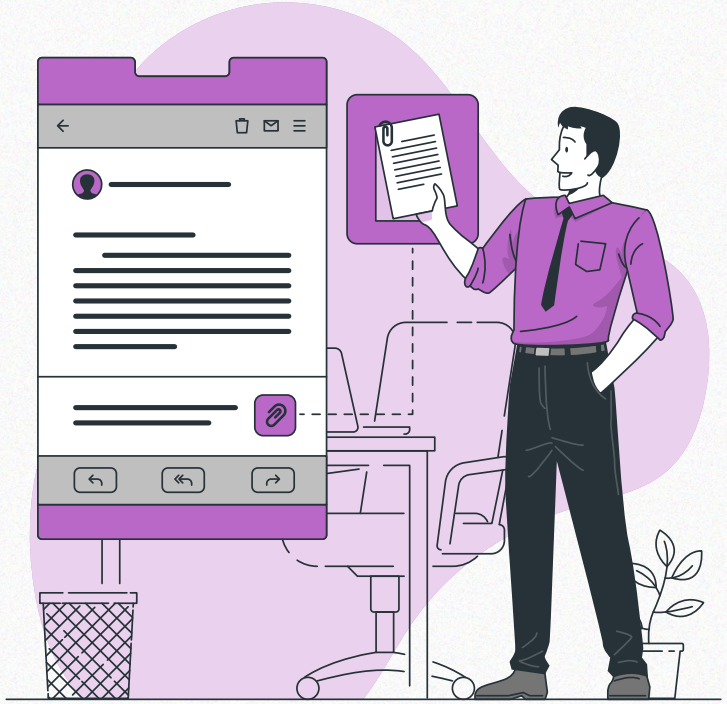
Pregunta reflexiva

¿Cómo diagnosticarías un problema de latencia intermitente que ocurre solo bajo condiciones muy específicas de alta concurrencia en un microservicio?

Ejemplo

En un escenario de microservicios, cada solicitud del usuario viaja por varios componentes: pasarela de pagos, servicio de inventario, servicio de facturación, etc.

Con *distributed tracing*, podemos ver si la demora ocurre en la confirmación de pago, en la gestión de stock o en la base de datos, y no solo “que hay un problema”, sino el punto exacto y el tipo de error.

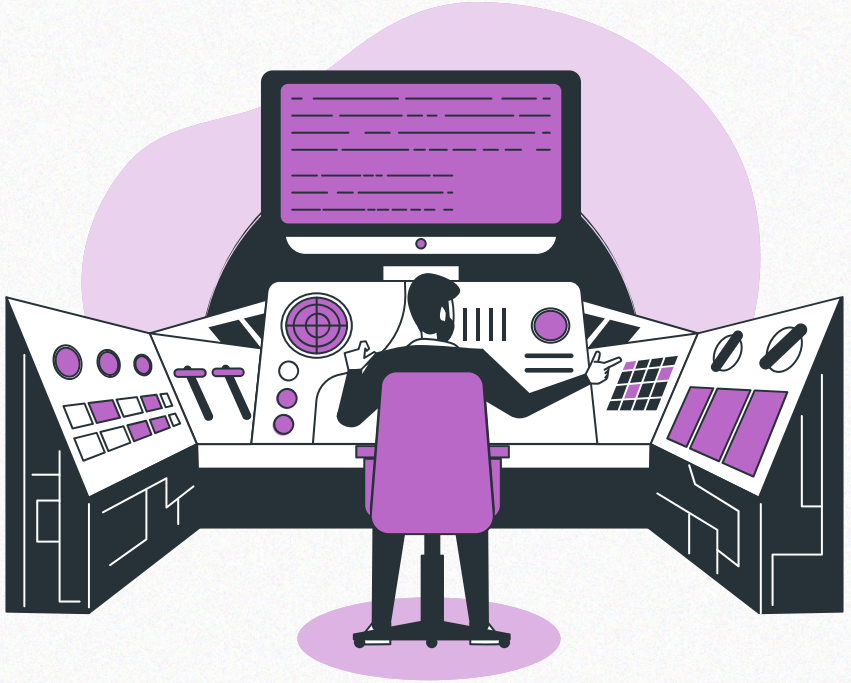


Convergencia de Monitoreo y Observabilidad

Si bien **monitoreo** y **observabilidad** son distintos en alcance, se complementan:

• El **Monitoreo** se enfoca en lo que sabemos que puede fallar (métricas definidas, umbrales).

• La **Observabilidad** busca darnos respuestas cuando no sabemos qué preguntar (eventos inesperados, patrones de error poco frecuentes).



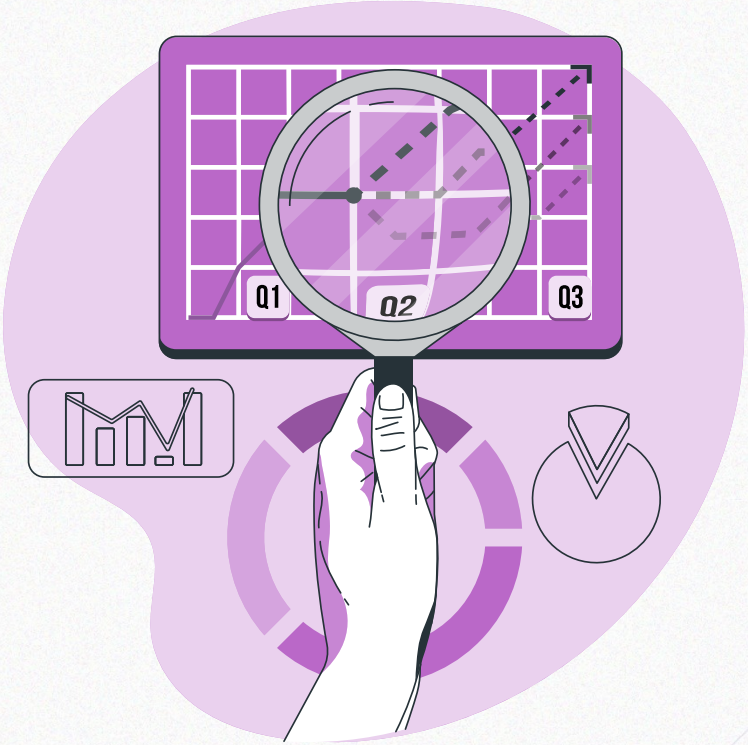
Software Architecture Metrics

Si mañana les pidieran demostrar con datos que su arquitectura es buena… ¿qué mostrarían?

- *“Qué miden hoy para decir que una arquitectura es buena?”*
- *“Son métricas técnicas? de negocio? o ambas?”*
- *“Cuánto de lo que miden está realmente vinculado con las decisiones arquitectónicas?”*

Software Architecture Metrics

Métricas diseñadas para evaluar atributos de calidad de una arquitectura, proporcionando evidencia objetiva para tomar decisiones y mejorar el diseño.



1. **Métricas estructurales**
   - Ej.: Acoplamiento, cohesión, complejidad ciclomática promedio por módulo.
   - *“Qué impacto tiene un acoplamiento alto en el lead time?”*
2. **Métricas de flujo y cambio**
   - Ej.: Time to market, frecuencia de despliegue, número de commits por módulo.
   - *“Cómo relacionar estos datos con la deuda técnica?”*
3. **Métricas de calidad operativa** (ligadas a la arquitectura)
   - Ej.: MTTR (Mean Time to Recovery), MTBF (Mean Time Between Failures).

Métricas puramente técnicas ≠ métricas útiles

Imaginemos que reducimos el tiempo de despliegue en un 50%. ¿Cómo demostramos que esto impacta en el negocio?

Qué ganaría el negocio si ustedes hoy bajaran su MTTR de 2 horas a 20 minutos?

- **Métrica técnica aislada:** "Acoplamiento promedio de 0.35" → no dice nada a un director comercial.
- **Métrica correlacionada:** "Reducir el acoplamiento de 0.5 a 0.35 bajó el tiempo promedio para lanzar nuevas funcionalidades de 3 semanas a 1, aumentando el *Monthly Active Users* en un 8%."

Tipo de métrica técnica (arquitectura)	**Métrica de negocio correlacionable**
MTTR (Mean Time To Recovery)	Pérdidas evitadas por downtime (USD)
Lead Time for Changes	Time-to-Market de nuevas funcionalidades
Acoplamiento entre módulos	Tiempo de entrega de features cross-team
Tasa de fallos por despliegue	Churn rate (clientes que se van por mala experiencia)
Cobertura de trazas y logging	Reducción de tickets de soporte o tiempo de resolución
Disponibilidad de servicios críticos	Conversion rate, ventas completadas

Caso de Ejemplo

Una plataforma SaaS redujo su *Change Failure Rate* de 15% a 5% gracias a refactorizar APIs y mejorar pruebas de integración.

Esto, combinado con una mejora de *Deployment Frequency*, permitió lanzar 3 features clave antes de la competencia.

Impacto medible: Aumento del 12% en *Monthly Active Users* y +7% en ingresos recurrentes.

Integración de la Evaluación Arquitectónica con Monitoreo y Observabilidad

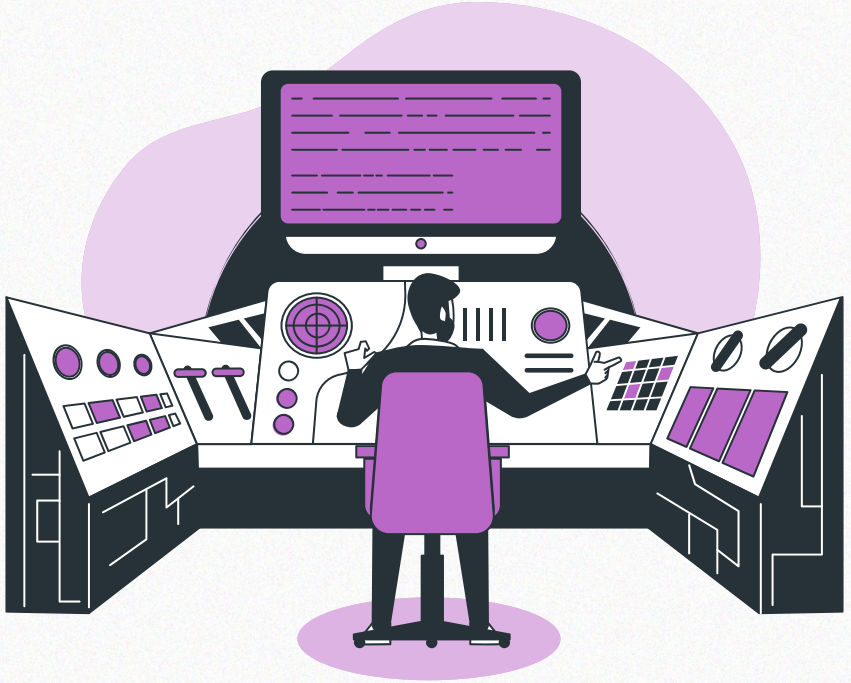
Si ATAM nos dice qué atributos importan y bajo qué escenarios sufren… ¿cómo se ve eso en producción el lunes?

Definir Atributos de Calidad y Establecer SLIs y SLOs

- Si ATAM indica que la **latencia** es crítica, se convierte en un **Service Level Indicator (SLI)**.

- Se define un **Service Level Objective (SLO)**, por ejemplo: “El 95% de las peticiones deben completarse en menos de 300 ms”.

- Esto se implementa a nivel de monitoreo mediante herramientas como Prometheus/Grafana para medir el tiempo de respuesta y generar alertas si se rompe la SLO.



Traducir ATAM a SLIs

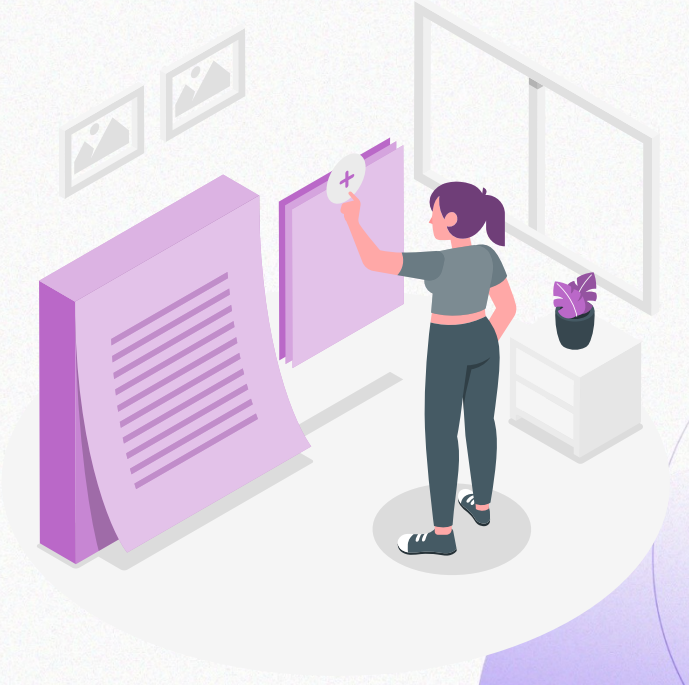
Escenario ATAM: *[fuente del estímulo] causa [estimulus] en [artefacto] bajo [entorno]; respuesta [medida] debe ser [umbral].*

- **Fuente del estímulo:** Usuario final en un evento de alto tráfico
- **Estímulo:** Realiza una compra en el sitio web
- **Artefacto:** Servicio de checkout
- **Entorno:** Carga 10× la media, con múltiples proveedores de pago activos
- **Respuesta:** El sistema procesa la transacción y responde
- **Medida:** Latencia de respuesta
- **Umbral:** < 300 ms para el 95% de solicitudes

SLI: *Porcentaje de solicitudes HTTP a /checkout cuya latencia p95 es menor a 300 ms.*

Definición de medición:

- **Fuente:** histogramas de latencia instrumentados con OpenTelemetry.
- **Unidad:** segundos.
- **Granularidad:** 5 min.
- **Segmentación:** country, payment_provider, device_type.



Del SLI al SLO

SLI: *Porcentaje de solicitudes HTTP a /checkout cuya latencia p95 es menor a 300 ms.*

Si ya tenemos el SLI definido, por qué no basta con medirlo y ya?

Un SLO es el objetivo cuantitativo que define el nivel esperado de una métrica (SLI) durante un periodo de tiempo determinado

Si el SLI mide *% de requests rápidas*, el SLO sería algo como: ***"El 95% de solicitudes a /checkout deben completarse en menos de 300 ms en cualquier ventana de 30 días."***

Elegir el umbral y la ventana de tiempo

Preguntas:

1. *"Si pongo el umbral muy alto (99.999%), qué pasará con la operación?"*
   - "Costos altos, estrés en equipos, alertas constantes".
2. *"Si pongo el umbral muy bajo (80%), qué pasará con el usuario?"*
   - "Mala experiencia, pérdida de negocio".
3. *"Por qué no usar solo promedios?"*
   - "Ocultan degradaciones importantes (picos lentos)".

Reglas prácticas:

- El umbral debe ser **ambicioso pero alcanzable**.
- Usar percentiles (p95/p99) en lugar de promedios.
- La ventana debe alinearse a los ciclos de revisión del negocio (semanal, mensual, trimestral).

Error Budget

El **error budget** es la cantidad de fallos que tu sistema puede permitirse antes de incumplir el SLO.

Se expresa como el **complemento** del SLO:

- SLO 95% → Error Budget = 5% de fallos tolerados.
- SLO 99.9% → Error Budget = 0.1% de fallos tolerados.

Ese 5% se puede traducir a:

- **Tiempo:** en 30 días (~720 horas), 5% = **36 horas** de incumplimiento.
- **Transacciones:** si procesamos 1 millón de solicitudes/mes, 5% = **50,000 solicitudes** pueden exceder los 300 ms.

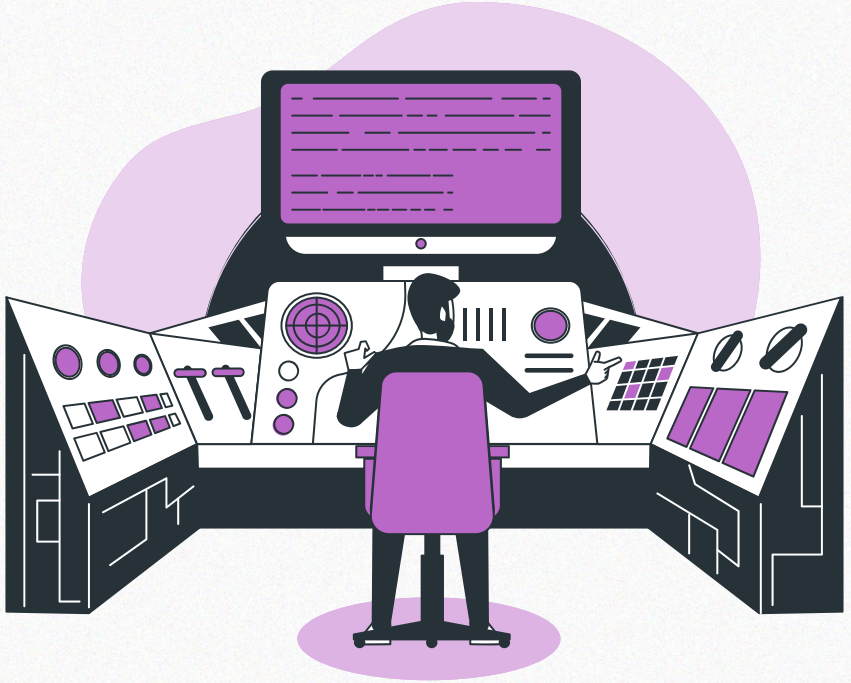
Importante: “Fuera de SLO” no siempre significa fallo total; puede ser lentitud, error HTTP, degradación parcial, etc., según cómo definiste el SLI.

Paso	Pregunta	Resultado
1. Definir SLI	“Qué medimos realmente?”	% de requests rápidas
2. Poner objetivo	“Cuál es un nivel aceptable?”	95% en < 300 ms
3. Fijar ventana	“En qué periodo evaluamos?”	30 días móviles
4. Calcular budget	“Qué margen de fallo aceptamos?”	5% (36h)
5. Política	“Qué hacemos si nos acercamos al límite?”	Alertar + pausar despliegues

Identificar Riesgos y Tradeoffs y Diseñar Instrumentación y Pruebas de Carga

- Si la arquitectura presenta un riesgo de dependencia en un módulo crítico (p. ej. un microservicio “gateway”), se instrumenta con mayor detalle ese componente (logs más detallados, métricas específicas, *distributed tracing*).

- En pruebas de carga (Stress, Soak, Spike Testing), se evalúa cómo se comportan estos módulos y se recogen datos de observabilidad para diagnosticar cuellos de botella.



Métricas de Arquitectura y Correlación con Datos de Observabilidad

- **Complejidad Ciclomática**: Indica la complejidad del flujo de control dentro de un módulo o servicio (cuántos caminos lógicos existen). *Cuanto mayor sea, más propenso puede ser el código a errores o dificultades de mantenimiento.*
- **Profundidad de Dependencias (Dependency Depth)**: Identifica cuán anidado está un componente dentro de la arquitectura. *Un sistema con dependencias profundas puede ser más complejo de monitorear y diagnosticar.*
- **Frecuencia de Cambios (Change Frequency)**: Con qué frecuencia cambia un módulo. *Módulos con cambios muy frecuentes pueden requerir especial atención en su monitoreo para detectar regresiones o errores tempranos.*

Ejemplo de Análisis de Métricas

Un sistema de microservicios de logística (*envíos*, *tracking*, *notificaciones a clientes*, *etc.*)

Descubrimiento: El servicio de “tracking” tiene alta complejidad ciclomática y una frecuencia de cambios muy elevada en cada sprint.

Acción: Se decide mejorar la observabilidad agregando *distributed tracing*, logs detallados y métricas de rendimiento específicas de las rutas críticas (*p.ej.*, *actualización masiva de estados de envío*).

Perspectivas de Operabilidad y Configuración de Dashboards y Alertas

Rozanski y Woods, en *Software Systems Architecture*, proponen **viewpoints** y **perspectives** para estructurar la arquitectura en capas de preocupación:

Viewpoints:

- **Vista Lógica**: Estructura y organización interna (módulos, clases, componentes).
- **Vista de Procesos**: Comportamiento en tiempo de ejecución, concurrencia e hilos de ejecución.
- **Vista de Implementación**: Cómo se gestiona el código, la configuración, la integración continua, etc.
- **Vista de Despliegue**: Cómo se despliega la solución en entornos físicos o virtuales, servidores, contenedores, etc.

Perspectivas de Operabilidad y Configuración de Dashboards y Alertas

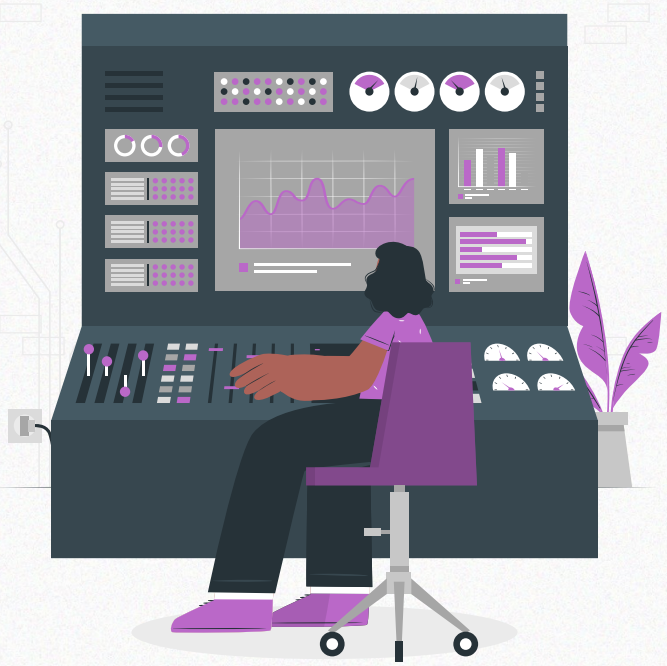
Rozanski y Woods, en *Software Systems Architecture*, proponen **viewpoints** y **perspectives** para estructurar la arquitectura en capas de preocupación:

Perspectives:

Seguridad, **rendimiento**, **mantenibilidad**, **disponibilidad**, **implementación y operatividad**, entre otras.

La perspectiva de **operabilidad** y **monitorización** es fundamental para planificar la **observabilidad**. Debe responder preguntas como:

- ¿Qué necesitamos medir?
- ¿Dónde y cómo registramos eventos importantes (logs)?
- ¿Cómo visualizamos y alertamos anomalías?



Estrategias y Herramientas Prácticas de Monitoreo y Observabilidad

Instrumentación y Recolección de Datos

La instrumentación es el proceso de **agregar "puntos de medición"** en el software o la infraestructura para recopilar datos relevantes. Estos datos pueden ser métricas, logs y trazas.

Requerimientos:

1. Recolectar métricas de latencia en el servicio de cobro (p.ej., "time to authorize payment").
2. Logs estructurados con la trazabilidad de cada transacción.
3. Alertas ante tasas de error > 1% en un intervalo de 5 minutos.

Instrumentación y Recolección de Datos

La instrumentación es el proceso de **agregar "puntos de medición"** en el software o la infraestructura para recopilar datos relevantes. Estos datos pueden ser métricas, logs y trazas.

Logs

Registros secuenciales que documentan eventos en el sistema, útiles para comprender la historia de la ejecución.

Buenas prácticas:

- Utilizar **logs estructurados** (JSON) que incluyan campos relevantes (timestamp, severity, mensaje, ID de la solicitud, etc.).
- Aplicar una política de retención y rotación: no es necesario guardar logs por tiempo indefinido; establecer reglas de archivado.
- Filtrar la información sensible (p. ej., datos personales) y cifrarla si es necesario.

Ejemplo:
En un microservicio de facturación, un evento de "Pago aprobado" incluiría campos como: { "level": "INFO", "orderId": "1234", "paymentStatus": "approved", "timestamp": "...", ... }.

Instrumentación y Recolección de Datos

La instrumentación es el proceso de **agregar "puntos de medición"** en el software o la infraestructura para recopilar datos relevantes. Estos datos pueden ser métricas, logs y trazas.

Métricas

Datos cuantitativos que describen el rendimiento o estado de un servicio (ej., latencia, uso de CPU, throughput).

Tipos de métricas

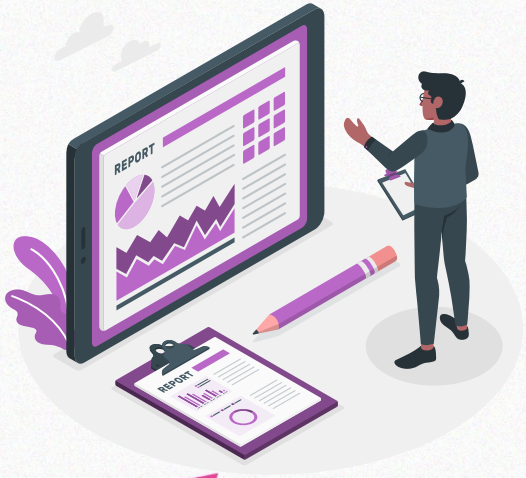
1. **Métricas de recurso**: CPU, memoria, espacio en disco.
2. **Métricas de rendimiento**: Latencia de respuesta, cantidad de peticiones por segundo (RPS), tasa de errores.
3. **Métricas de saturación**: Uso de CPU por encima de cierto umbral, conexiones de base de datos al máximo, etc.

Herramientas populares:

• **Prometheus**: Sistema de monitoreo y *time-series database* (TSDB). Permite etiquetar métricas e implementar consultas y alertas.
• **Grafana**: Plataforma de visualización que se integra con Prometheus, InfluxDB, Elasticsearch, etc.

Instrumentación y Recolección de Datos

La instrumentación es el proceso de **agregar "puntos de medición"** en el software o la infraestructura para recopilar datos relevantes. Estos datos pueden ser métricas, logs y trazas.



Trazas Distribuidas

Permite seguir el recorrido de una petición a través de múltiples microservicios, mostrando cuánto tiempo pasa en cada uno.

Importancia:

- Identificar cuellos de botella en sistemas complejos.
- Aislar las causas de la latencia en un flujo con múltiples servicios (base de datos, caché, APIs externas, etc.).

Herramientas:

- **Jaeger** y **Zipkin**: Soluciones populares de *distributed tracing*.
- **OpenTelemetry**: Estándar abierto para la instrumentación de métricas, logs y trazas; facilita la recolección y exportación a distintos backends.

Modos de Instrumentación

Instrumentación Automática

Librerías o agentes que interceptan llamadas (HTTP, DB, mensajería, etc.) sin necesidad de escribir código manual.

Instrumentación Manual

El desarrollador escribe llamadas explícitas a las APIs de logging, métricas o tracing en puntos concretos del código.

```
from prometheus_client import

Counterpayment_requests =
Counter('payment_requests_total', 'Número de
peticiones de pago procesadas’)

def process_payment(order_id, amount):
        payment_requests.inc()
        # ... lógica de pago ...
```

Distributed Tracing

Trace (Traza) Una *traza* representa el recorrido completo de una petición o “transacción” a través de varios servicios.

- Por ejemplo, si un usuario realiza una compra en un sitio de e-commerce, esa acción puede generar peticiones al servicio de catálogos, servicio de pagos, servicio de inventario y servicio de notificaciones. Todos esos pasos forman parte de la misma “traza”.

Span (Segmento) Un *span* es una unidad de trabajo dentro de la traza. Cada servicio o componente que procesa la petición genera uno o varios spans.

Cada span incluye información como:

- Nombre del servicio o la operación (ej. GET /payment/authorize)
- Timestamp de inicio y fin (para medir duración)
- Metadatos (o **tags**) que describen lo que ocurrió (por ejemplo, “status=200”, “userId=123”)
- misma traza.

Trace ID y Span ID Para enlazar todo adecuadamente, cada traza tiene un **Trace ID** único que se transmite entre servicios. Dentro de cada traza, cada span tiene un **Span ID** único.

- Si un servicio “A” llama a un servicio “B”, el span del servicio A es el *padre*, y el span del servicio B es el *hijo*; el servicio A envía al B el Trace ID para que B sepa que ambos pertenecen a la

Context Propagation (Propagación de Contexto) Para que cada microservicio sepa a cuál traza pertenece la petición, se deben pasar ciertos encabezados (headers) o metadatos en la llamada HTTP/RPC.

- Ej.: traceparent (en OpenTelemetry) o x-b3-traceid (en Zipkin).
- Al recibir la petición, el servicio hijo extrae esos encabezados y continúa la traza, agregando su propio span.

Flujo General de una Solicitud con Tracing

1

Servicio Frontend (Cliente web o API Gateway)

- El frontend recibe la solicitud del usuario (p.ej. un “checkout”).
- Se genera un **Trace ID** y un span inicial (span “A”).
- Se llama al siguiente servicio, incluyendo el **Trace ID** y el **Span ID** en las cabeceras de la petición.

```
POST /payments/authorize HTTP/1.1
Host: payments.internal
traceparent: 00-
4bf92f3577b34da6a3ce929d0e0e4736-
3bd9f0d48f1b82da-01
Content-Type: application/json


{ "order_id": "XYZ", "amount": 59.99 }
```

traceparent: 00-<TraceID>-<ParentSpanID>-<traceFlags>

2

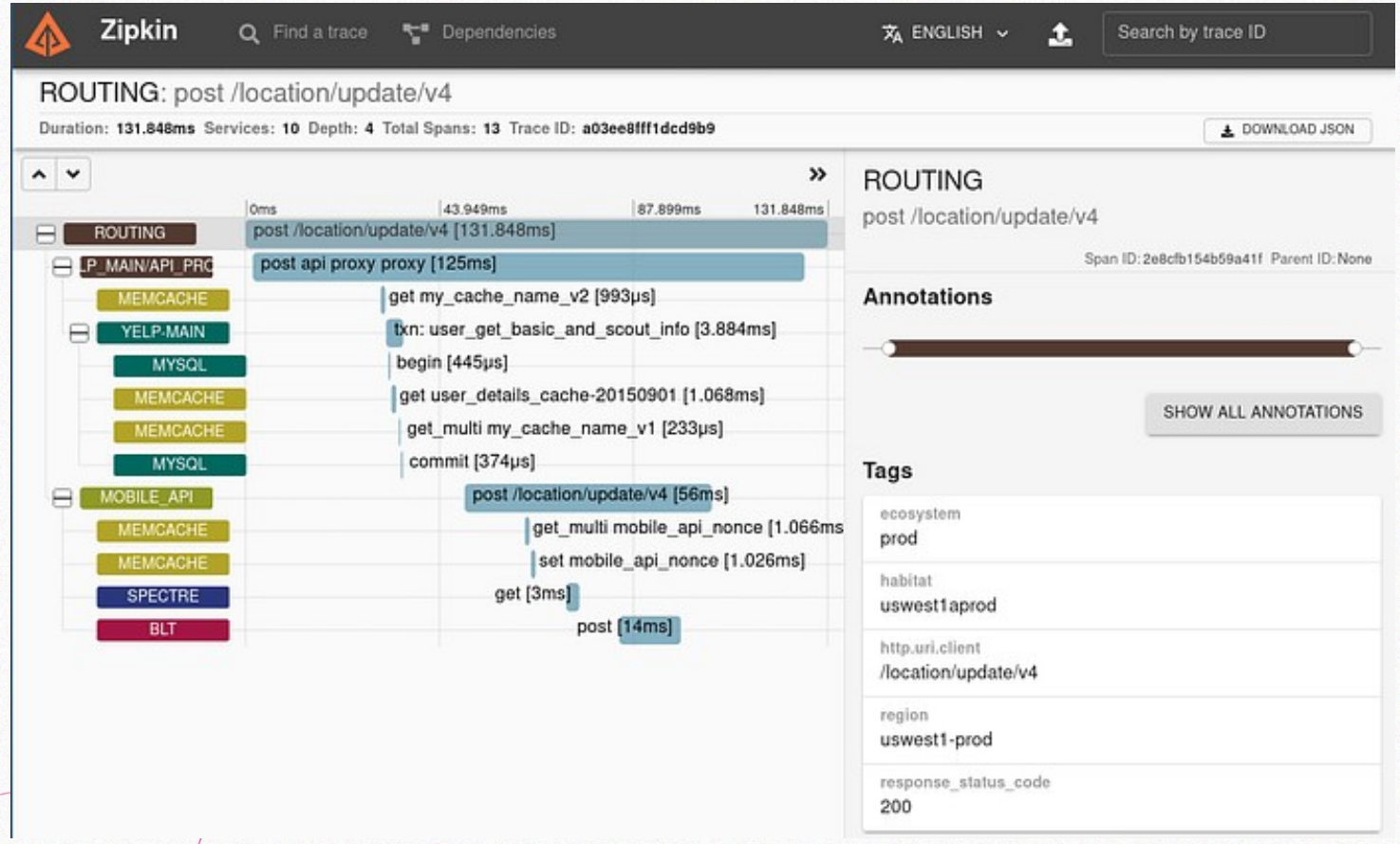
Servicio de Pagos

- Recibe la llamada y detecta los encabezados de tracing.
- Crea su propio span (span “B”), indicando que es “hijo” del span “A”.
- Registra metadatos: tiempo de inicio, tags relevantes (montos, tipo de tarjeta, etc.).
- Después de autorizar el pago, llama a un servicio externo de cobro (tercero) o a la base de datos, continuando con la misma lógica de propagación.

```
Trace ID: 4bf92f3577b34da6a3ce929d0e0e4736

Span (A) - Checkout_Initiate (frontend)
  └─ Span (B) - Authorize_Payment (payments-service)
        └─ Span (B1) - ExternalPaymentCall (pasarela ex
  └─ Span (C) - Check_Stock (inventory-service)
  └─ Span (D) - Send_Notification (notifications-servi
```

Herramientas de Tracing



JAEGER

ZIPKIN

Diseño de Tableros y Alertas

Diseño de Tableros (Dashboards) y Alertas

Un aspecto fundamental de la **observabilidad** es contar con **tableros (dashboards) y sistemas de alertas** que presenten la información de forma clara, accionable y alineada con los objetivos de negocio y de confiabilidad del sistema.

No basta con recolectar métricas y trazas: es igual o más importante **visualizar** esos datos de manera que los equipos puedan tomar decisiones rápidamente y reaccionar ante problemas antes de que se vuelvan críticos.

Principios de Diseño de Dashboards

Jerarquía de Información

Un buen tablero prioriza las **métricas más importantes** en la parte superior o central, de forma que se vean en un solo vistazo.

Métricas secundarias o de diagnóstico más detallado van en secciones inferiores o en “sub-tableros” (dashboards hijos) para exploración más profunda.

Principios de Diseño de Dashboards

Visibilidad de los KPIs / SLIs más Críticos

Identificar los **indicadores claves** que reflejan la salud del sistema, a menudo ligados a los SLOs definidos (por ejemplo, latencia p90 < 300ms, tasa de error < 1%, disponibilidad del 99.9%).

Resaltar estos KPIs con gráficos claros (gráficas de línea o paneles numéricos).

Principios de Diseño de Dashboards

Simplicidad y Claridad en Visuales

Evitar llenar la pantalla con demasiadas gráficas. El exceso de paneles o colores produce confusión y *fatiga visual*.

Usar un lenguaje visual consistente: escala de colores para niveles de severidad (verde/amarillo/rojo), estilos de gráfico para cada tipo de métrica (por ejemplo, barras para conteos, líneas para series temporales, *heatmaps* para densidad, etc.).

Principios de Diseño de Dashboards

Contexto y Segmentación

Incluir **filtros** que permitan segmentar por región, versión del sistema, cliente, etc.

Esto ayuda a responder preguntas como “¿este pico de latencia ocurre en todos los data centers o solo en uno?”.

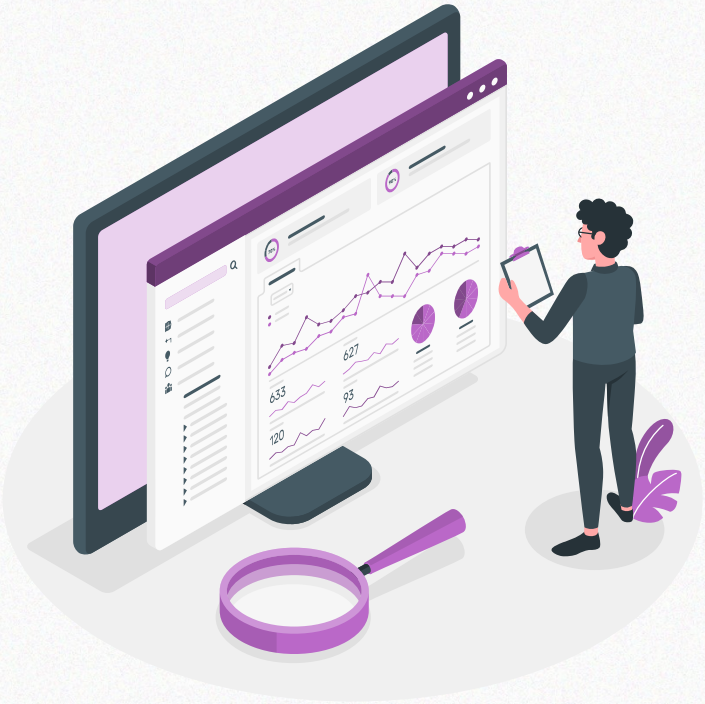
Poder “abordar el problema” de lo general a lo específico agiliza el diagnóstico.

Principios de Diseño de Dashboards

Metadatos y Alertas Integradas

Mostrar alarmas activas o eventos recientes (por ejemplo, un despliegue que coincide con un aumento de errores) en el mismo tablero.

Incluir enlaces a *runbooks* o documentación de incidentes para agilizar la acción correctiva.



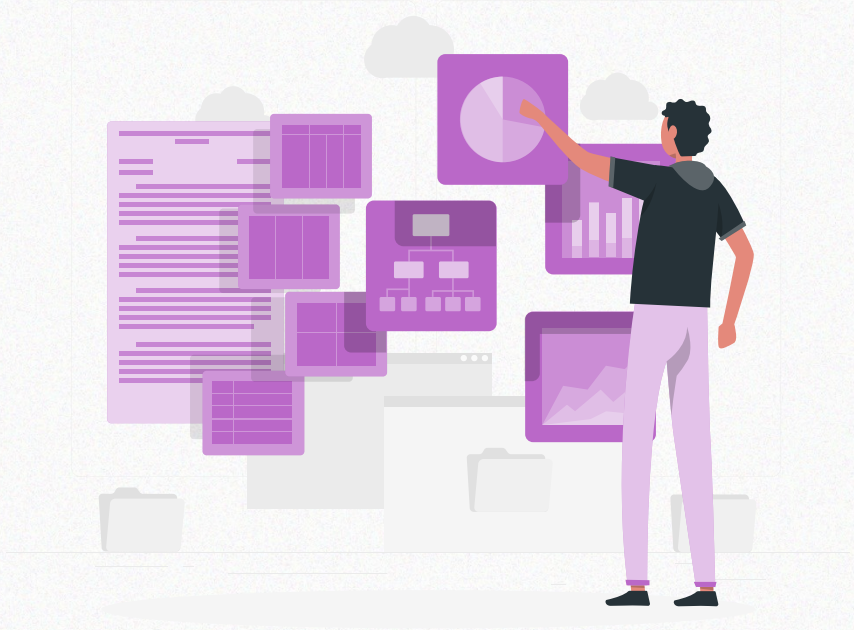
Principios de Diseño de Dashboards

Diseño para Diferentes Audiencias

Equipos de SRE/DevOps necesitan ver métricas técnicas (latencia, errores 5xx, CPU, memoria).

Stakeholders o gerencia pueden querer paneles con métricas de negocio (tasa de transacciones, ingresos, usuarios activos).

Evitar mezclar en exceso métricas de negocio y técnicas en el mismo tablero si generan confusión.



Estrategias para Construir Dashboards Efectivos

Utilizar Paneles Sintéticos y Paneles Detallados

- Un **panel sintético** agrupa en un solo vistazo las métricas clave (CPU, memoria, errores), ayudando a detectar tendencias generales.
- Luego, **paneles detallados** profundizan en cada servicio o métrica cuando necesitamos diagnosticar la raíz de un problema.

Estrategias para Construir Dashboards Efectivos

Crear Dashboards por Dominio o Funcionalidad

En un entorno de microservicios, cada servicio puede tener su propio dashboard que muestre sus métricas de rendimiento y disponibilidad.

- Un “Dashboard de Mensajería” podría mostrar cola de mensajes, tasa de consumo, retrasos en la entrega, etc.
- Un “Dashboard de Pagos” enfocado en latencia de autorización, tasas de abandono de carrito, etc.

Estrategias para Construir Dashboards Efectivos

Representar la Relación entre Servicios

Con herramientas de topología (por ejemplo, en Grafana, paneles de red o diagramas de flujo) podemos representar las dependencias y tiempos de respuesta entre microservicios.

Esto facilita entender que si el “Servicio A” se cae, puede impactar a “Servicio B” y “Servicio C”.

Estrategias para Construir Dashboards Efectivos

Comparaciones Temporales y Alertas Embebidas

Mostrar rangos de tiempo comparativos (“last 6h vs. same period last week”) para identificar comportamientos anómalos de estacionalidad o picos de tráfico.

Tener alertas integradas (por color, umbrales) en los gráficos mismos: si la latencia excede cierto valor, se resalta la línea en rojo.

Configuración de Alertas: Claves para Evitar Ruido y Falsos Positivos

Las alertas son un mecanismo que **notifica al equipo** cuando algo se sale de los límites esperados.

Sin embargo, **un sistema de alertas mal configurado** puede causar *alert fatigue*, bloqueando la efectividad del equipo.

Definir el Objeto de la Alerta

¿Qué es lo que realmente importa?

¿Que la latencia supere 300 ms?

¿Que el servicio no responda por 1 minuto?

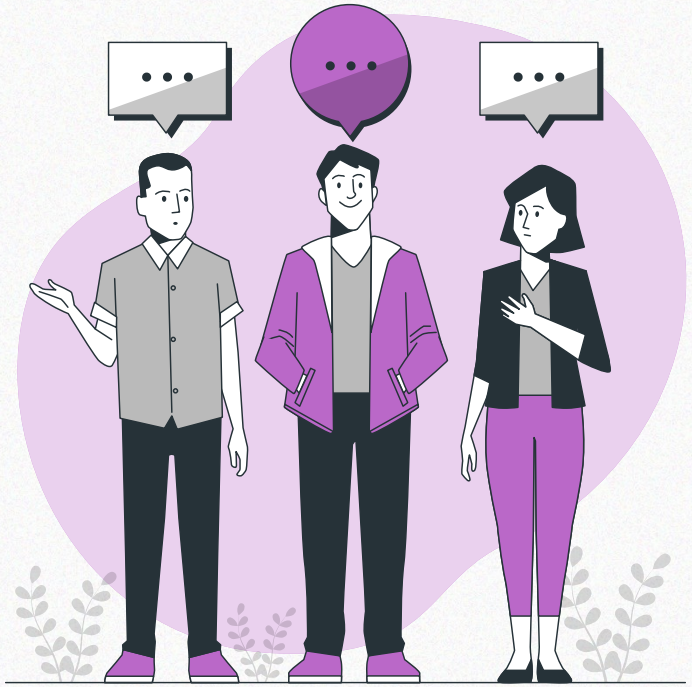
¿Que el uso de CPU sea crítico (90% sostenido por 5 minutos)?

Basarse en SLOs y SLIs

Una buena práctica, difundida por *Site Reliability Engineering (SRE)* de Google, es **definir SLOs (Service Level Objectives)**:

- Ej.: “El 95% de las peticiones deben tener una latencia < 200ms”.

Las alertas se disparan si vemos que **no** cumpliremos ese SLO.

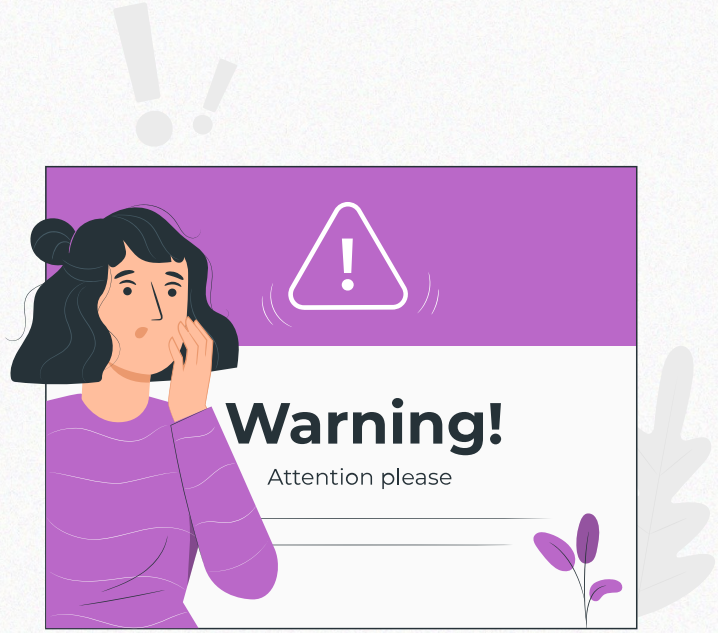


Alertas Escalonadas (Multi-Level Alerts)

Warning (Advertencia): El sistema se acerca a un umbral pero aún funciona. Puede que no requiera acción inmediata, pero sí seguimiento.

Critical: El sistema está realmente en peligro. Hay que reaccionar rápido.

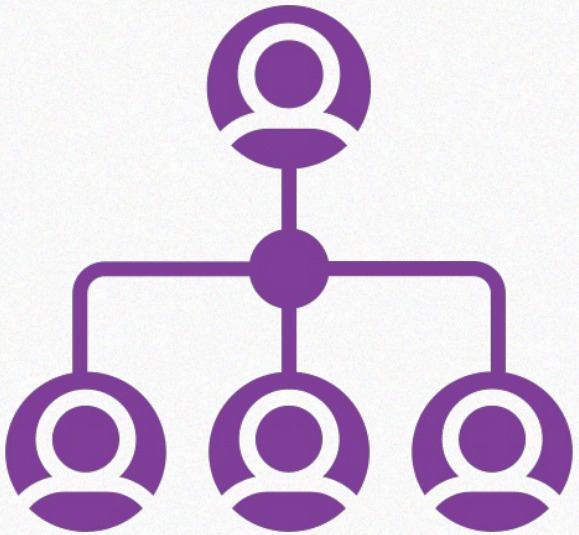
Diferenciar niveles ayuda a priorizar. Un uso de CPU del 80% quizás sea **Warning**; un 95% sostenido con colas de requests crecientes es **Critical**.



Incluir Contexto en la Alerta

La notificación no solo debe decir “latencia alta”, sino también “Servicio X supera latencia de 300ms por 5 minutos en la región Y, ver logs en ”.

Incluir **links a dashboards**, logs, documentación o *runbooks* (guías de resolución) acelera la respuesta a incidentes.



Alertar a las Personas Correctas

No todos los incidentes deben llegar a todo el equipo.

Usar canales adecuados (Slack, correo, SMS, PagerDuty) y definir escalamientos.

Ej.: “Si la alerta no se resuelve en 10 minutos, escalar al líder de SRE”.

Ejemplos de Configuración de Alertas

1. **Alerta por Tasa de Error**

***Condición*:** Tasa de errores 5xx > 2% en un periodo de 5 minutos.

***Acción*:** Enviar notificación a Slack “#oncall-sre” con: “Servicio X - Error rate 5xx excede 2% (actual: 3.5%)”.

***Link*:** Dashboard de Errores en Grafana con filtros predefinidos.

2. **Alerta por Latencia (SLO)**

***Definición de SLO*:** “95% de las peticiones con latencia < 300 ms”.

***Condición de alerta*:** Proyección indica que caeremos debajo de ese objetivo en las próximas 4 horas (basado en la tasa actual de incumplimiento).

***Notificación*:** “Latencia SLO en riesgo - 25% de error budget consumido en las últimas 2 horas”.

Ejemplos de Configuración de Alertas

3. Alerta por Saturación de Recursos

***Condición*:** CPU usage > 90% sostenido por 10 minutos + la cola de requests de CPU en la instancia es alta.

***Acción*:** Notificar al equipo de infraestructura y adjuntar un runbook con instrucciones: “Verificar si se necesita autoescalado o redistribución de cargas”.

4. Alerta de Disponibilidad (Heartbeat)

***Condición*:** No se han recibido *heartbeats* de un microservicio en 2 minutos.

***Acción*:** Notificar al equipo dev correspondiente “Servicio Inactivo”, para que revisen si ha sufrido un crash o problemas de red.

Buenas Prácticas para el Mantenimiento

Revisar y Depurar Alertas Frecuentemente: Eliminar alertas obsoletas, fusionar duplicadas, ajustar umbrales según la realidad del sistema.

Analizar Post-Incidencia (Postmortem): Después de un incidente, preguntarse “¿Funcionó nuestra alerta? ¿Fue oportuna? ¿Faltó información en el dashboard?” y aplicar mejoras.

Documentar: Mantener guías de uso de cada dashboard y de resolución de incidentes en un repositorio accesible al equipo.

Automatización e Integración con CI/CD

Automatización e Integración con CI/CD

La **automatización** y la **integración continua/despliegue continuo (CI/CD)** son prácticas esenciales para garantizar que nuestro sistema no solo se mantenga funcional y libre de regresiones, sino también **monitoreado y observable** a lo largo de su ciclo de vida.

El objetivo es que cada nuevo despliegue cuente con la instrumentación, métricas y alertas actualizadas, y que podamos detectar fallas o anomalías de manera temprana, incluso antes de llegar a producción o en un entorno controlado.

Pruebas de Monitoreo y Observabilidad en la Cadena de CI/CD

Validación de Instrumentación en el Entorno de Pruebas

Cuando se hace un *build* de la aplicación, se puede **verificar** que las librerías de instrumentación (tracing, logs, métricas) estén correctamente configuradas y no generen errores de compilación o runtime.

Se pueden incluir **pruebas unitarias/integrares** que confirmen la existencia de endpoints de métricas (p. ej., /metrics en Prometheus), o que ciertos logs se generen adecuadamente en escenarios controlados.

Pruebas de Monitoreo y Observabilidad en la Cadena de CI/CD

Pruebas de Carga y Rendimiento (Performance Tests)

Después de desplegar en un entorno de *staging* (o preproducción), se ejecutan pruebas de rendimiento con herramientas como **k6, Gatling o JMeter**.

- Se monitorean las métricas (CPU, memoria, latencia) durante la prueba para ver si los **SLOs** se cumplen.
- Si no se cumplen los umbrales de latencia o tasa de error, se falla la *pipeline*, evitando que el código llegue a producción.

Pruebas de Monitoreo y Observabilidad en la Cadena de CI/CD

Pruebas de Contrato con el Sistema de Monitoreo

Asegurarse de que los dashboards y alertas asociados a ese servicio estén actualizados y sigan funcionando (por ejemplo, si cambió el nombre de una métrica o endpoint).

Verificar que las *alertas* puedan dispararse: simular escenarios de error y comprobar que el sistema de alertas (Prometheus Alertmanager, Datadog, etc.) notifique correctamente.

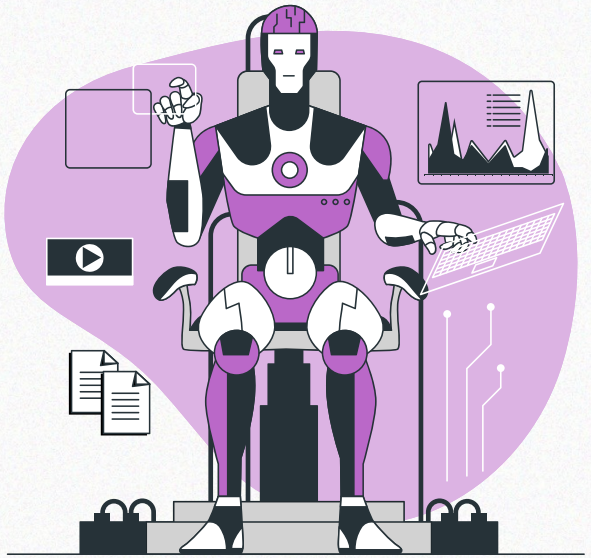
Ejemplo:

1. **Stage** “build”: compila la aplicación con la librería de OpenTelemetry.
2. **Stage** “test”: corre pruebas unitarias y validación de endpoints de monitoreo (p. ej., GET /metrics debe retornar 200 OK).
3. **Stage** “staging-deploy”: despliega en un entorno de staging y ejecuta pruebas de carga (k6).
4. **Stage** “staging-validation”: analiza las métricas recolectadas. Si la latencia p95 > 300ms, falla el pipeline.
5. Si todo pasa, se procede a producción.

Observabilidad como Código (OaC)

Observabilidad como Código (OaC) es un enfoque que extiende los principios de **Infraestructura como Código (IaC)** y de la filosofía "*Everything as Code*" a los **componentes de observabilidad**.

Esto incluye la definición y versionado de **métricas, dashboards, alertas, reglas de monitoreo, configuraciones de tracing** e incluso la propia infraestructura de recolección (collectors, exporters, etc.) mediante archivos de configuración o scripts automatizados.



¿Por qué Observabilidad como Código?

1. **Versionado y Auditoría**
• Al almacenar toda la configuración de observabilidad (dashboards, alertas, flujos de recolección) en un repositorio de código (Git, por ejemplo), cada cambio queda registrado con su autor, fecha y justificación.
• Permite hacer *rollback* si un cambio en las alertas o dashboards introduce un problema.

2. **Revisión por Pares (Pull Requests)**
• Cualquier modificación a una alerta, métrica o panel se somete al mismo proceso de *code review* que el resto del software, garantizando calidad y evitando configuraciones accidentales.

3. **Despliegue Consistente**
• Un pipeline de CI/CD aplica la configuración de observabilidad de manera automática en entornos de **staging** o **producción**, asegurando que no haya discrepancias manuales (la famosa frase “funcionaba en mi máquina” también puede pasar en monitoreo).

4. **Facilidad de Replicación de Entornos**
• Cuando se crean nuevos entornos (nuevas regiones, entornos de test), se despliega la misma configuración de dashboards, alertas y reglas de monitoreo con un simple comando (por ejemplo, terraform apply).

5. **Documentación Implícita**
• El código en sí mismo describe la configuración de la observabilidad. Leer un archivo YAML o HCL revela cómo están definidas las alertas, qué métricas se recolectan y cómo se visualizan, reduciendo ambigüedades.

Componentes de Observabilidad Versionables

1. **Configuraciones de Recolección e Instrumentación**
• Scripts para instalar y configurar agentes (Prometheus, Datadog, Elastic APM, OpenTelemetry Collector).
• Enfoque "*agent-as-code*": dónde se despliegan, qué puertos escuchan, qué endpoints scrapean, etc.

2. **Dashboards**
• Archivos JSON/YAML (por ejemplo, en **Grafana** se puede exportar o definir dashboards como JSON).
• Se definen paneles, tipos de gráfico, queries, variables y se almacenan en el repositorio.

3. **Alertas y Regla de Monitoreo**
• Archivos YAML (Prometheus Alertmanager), configuración de Datadog monitors, Splunk alerts o similares.
• Indican *thresholds*, acciones de notificación, niveles de severidad y quién debe recibir la alerta.

4. **Configuraciones de Trazas (Tracing)**
• OpenTelemetry Collector: pipelines de procesamiento, *exporters*, *receivers* y *processors* definidos en YAML.
• Políticas de *sampling*, máscaras de datos sensibles, etc.

5. **Scripts o Templates de Integración**
• Para inyectar variables de entorno, credenciales de acceso a backend de monitoreo, IDs de organización, etc., dependiendo del entorno (dev, staging, producción).

Herramientas y Metodologías para OaC

1. **Terraform**
• Módulos y *providers* específicos para **Grafana**, **Prometheus**, **Datadog**, **New Relic**, etc.
• Ejemplo: terraform apply genera dashboards, alertas y reglas de scraping en Prometheus, igual que crea máquinas virtuales o bases de datos.

2. **Grafana Provisioning**
• Permite cargar dashboards, datasources y *alerting rules* directamente desde archivos YAML/JSON en el arranque de Grafana.
• Estos archivos se incluyen en el repositorio y se versionan.

3. **Pulumi**
• Similar a Terraform, pero con soporte para lenguajes de programación (TypeScript, Python, Go, etc.).
• Permite definir la infraestructura y la configuración de observabilidad en código imperativo.

4. **Ansible/Chef/Puppet**
• Roles/recipes para instalar y configurar servicios de monitoreo (Prometheus, Alertmanager, Jaeger, etc.) y cargar sus archivos de configuración.

5. **GitOps**
• Filosofía en la que el repositorio Git es la “fuente de la verdad” y un *operator* aplica los cambios automáticamente en Kubernetes o en la infraestructura declarada.
• Se usa mucho con Argo CD, Flux y otras herramientas que “reconcilian” el estado deseado (definido en el repo) con el estado real.

Flujo Típico de Desarrollo con OaC

1. **Desarrollador Necesita una Nueva Métrica o Alerta**
• Se actualiza el código de la aplicación para exponer la nueva métrica o se define en la librería de tracing.
• Simultáneamente, en el mismo *branch*, se modifica el archivo YAML o JSON que define el dashboard y/o la regla de alerta.
• Se hace un pull request con ambos cambios.

2. **Revisión de Cambios (Pull Request)**
• El equipo revisa: “¿Tiene sentido la nueva métrica ‘billing_invoices_total’? ¿Es correcto el umbral de alerta a 5% de error?”
• Si está todo bien, se aprueba y se fusiona en la rama principal.

3. **Pipeline de CI**
• Valida sintaxis (p. ej., promtool check rules) para las alertas, verifica la estructura de los JSON de Grafana.
• Corre pruebas unitarias de la aplicación para asegurarse de que la métrica se expone bien.

4. **Despliegue en Staging (CD)**
• Se ejecuta un terraform apply (o análogo) que crea/actualiza la configuración de dashboards y alertas en el entorno staging.
• La aplicación se despliega con la nueva métrica.
• Se validan manual o automáticamente que la alerta se dispare correctamente (inyección de error o pruebas de carga).

5. **Promoción a Producción**
• Mismo *pipeline*, a veces con un paso adicional de aprobación.
• Resulta en la misma configuración de observabilidad aplicada al entorno de producción.

Ejemplo de Observability as Code con Terraform y Grafana

Supongamos que tenemos un archivo grafana_dashboards.tf:

```
provider "grafana" {
  url  = "https://grafana.mycompany.com"
  auth = var.grafana_api_key
}

resource "grafana_dashboard" "service_billing" {
  config_json = file("${path.module}/dashboards/billing-dashboard.json")
  folder      = "BillingServices"

  # se pueden parametrizar variables con Terraform
}
```

billing-dashboard.json contiene toda la definición del dashboard, incluidos paneles, queries de Prometheus, alertas, etc.

En la **pipeline** de CI/CD, al aprobar el PR, se ejecuta:

```
terraform init
terraform apply -auto-approve
```

Y esto sube o actualiza el dashboard en Grafana de forma automática.

Ejemplo de Observability as Code con Terraform y Grafana

Para alertas en Prometheus se podría tener otro archivo prometheus_alerts.tf con:

```
resource "prometheus_alert_rule" "billing_high_error_rate" {
  rule_group = "billing"
  namespace  = "default"


  rule {
    alert       = "BillingHighErrorRate"
    expr        = "sum(rate(billing_service_errors_total[5m])) /
sum(rate(billing_service_requests_total[5m])) > 0.05"
    for         = "1m"
    labels      = { severity = "critical" }
    annotations = {
      summary = "High error rate in billing service"
      description = "More than 5% errors in the last 5 minutes."
    }
  }
}
```

Cada cambio en este archivo se controla en Git, se revisa y se despliega automáticamente.

Buenas Prácticas en Observability as Code

1. **Desacoplar Configuraciones por Servicio/Dominio**
• Facilita que cada equipo o *squad* administre sus propios dashboards y alertas sin pisarse entre sí.
• Estructurar el repositorio con carpetas por microservicio o componente.

2. **Uso de Módulos/Reutilización**
• Definir “plantillas” de dashboards o sets de alertas que se puedan reutilizar en varios servicios, evitando duplicar definiciones.
• Ej.: un módulo para microservicios HTTP que incluye paneles de latencia, RPS y tasa de error por defecto.

3. **Testing y Validación**
• Integrar herramientas de lint (p. ej., promtool, terraform validate, grafana-check-dashboards si existe) en la pipeline.
• Probar localmente en un entorno de prueba antes de fusionar al repo principal.

4. **Mantener la Documentación Actualizada**
• Explicar en README o en archivos Markdown en el repo, qué significan las métricas y alertas definidas, guías de solución y canales de escalamiento.
• Asignar “dueños” de dashboards y alertas para garantizar mantenimiento continuo.

5. **Apuntar a un Modelo GitOps**
• Si se usa Kubernetes, se pueden emplear operadores como **Flux** o **Argo CD** que supervisan el repositorio y aplican cambios automáticamente.
• Así, la infraestructura de observabilidad está siempre “en sintonía” con el repositorio Git.